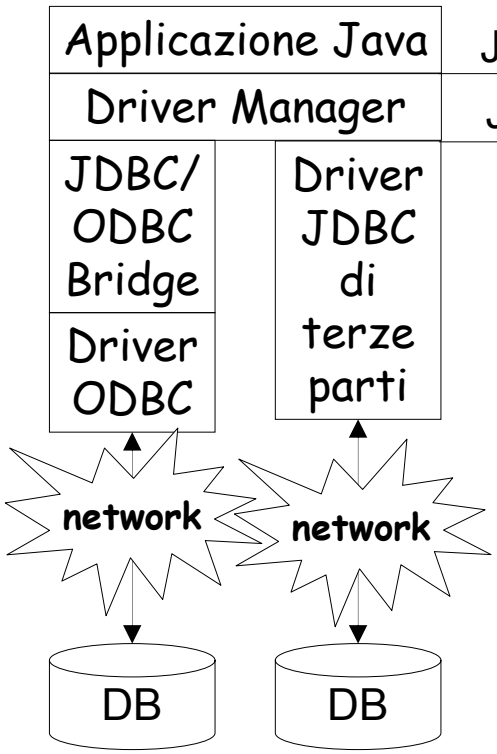
Linguaggio Java
JDBC

- L'architettura JDBC: driver, connessione, statement e result set
- Invio di comandi SQL
- Transazioni: COMMIT, ROLLBACK e livelli di isolamento
- Utilizzo avanzato dei result set: cenni
- Connessione a: fonti di dati ODBC, MySQL

Architettura di JDBC



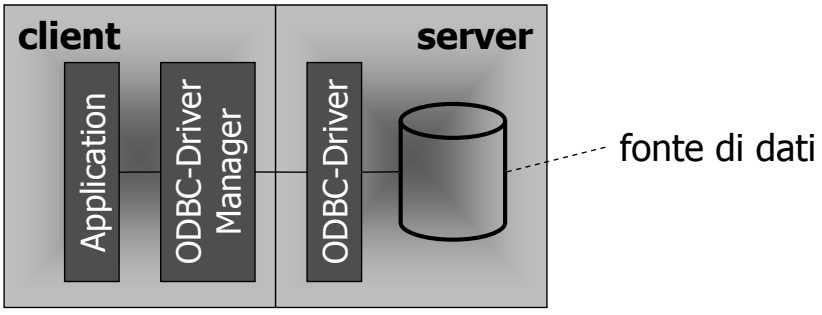
- Un applicazione interagisce con il **Driver Manager**, attraverso le JDBC API, inviandogli comandi SQL
- Il Driver Manager interagisce con i driver forniti da terze parti che effettivamente si collegano ai database e ritornano i risultati prodotti dai comandi SQL
- Il protocollo tra il driver ed il DB è proprietario

Tipi di JDBC driver

- **Driver di Tipo 1**: **traduce JDBC in ODBC**; assume che il driver ODBC sia presente e configurato correttamente
- **Driver di Tipo 2**: sviluppato **parzialmente in Java e parzialmente in codice nativo** (che comunica con il DB); richiede l 'installazione di librerie specifiche per la piattaforma
- **Driver di Tipo 3**: è una **libreria cliente in puro Java** che comunica con una componente server proprietaria; la componente client è indipendente dal database
- **Driver di Tipo 4**: è una **libreria in puro Java** che traduce i comandi JDBC direttamente in un protocollo proprietario

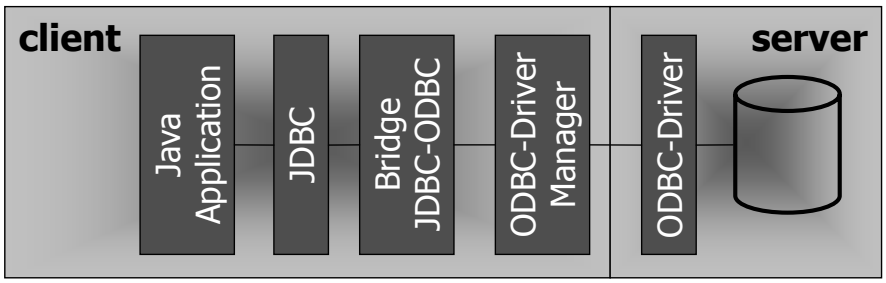
Connessione ad una fonte di dati ODBC

- ODBC è un protocollo Microsoft finalizzato alla comunicazione tra un applicazione ed una **fonte di dati**
  - Esempio di fonte dati: foglio Excel, un database, un documento Word
- ODBC mette a disposizione del programmatore una **API** implementata da un **ODBC Driver** fornito dal produttore della fonte dati



Il Bridge JDBC-ODBC

- Quando Sun ha rilasciato JDBC, i driver ODBC erano già disponibili per molti fornitori, perciò SUN ha rilasciato un **bridge JDBC-ODBC**
- Una soluzione basata su bridge JDBC-ODBC è da considerarsi **transitoria** in attesa di un driver JDBC
- Il bridge JDBC-ODBC è realizzato dalla classe **sun.jdbc.odbc.JdbcOdbcdriver** disponibile con JDK (**jre\lib\rt.jar**)



JDBC vs ODBC

- **ODBC può essere utilizzato solo da applicazioni client che girano in ambiente Windows**. JDBC invece è multipiattaforma.
- **ODBC è difficile da imparare**
- **ODBC ha una filosofia opposta a quella di Java**: "pochi metodi complessi con tante opzioni" invece di "*tanti metodi semplici ed intuitivi*"
- **ODBC fa riferimento a void* del linguaggio C che sarebbe innaturale in Java**
- **I driver ODBC devono essere installati a "mano"** mentre i driver "pure Java" possono essere caricati remotamente

I Data Source Name (DSN)

- Per accedere ad una fonte di dati ODBC è necessario memorizzare, nel **registro di configurazione** di Windows, i **parametri** necessari alla connessione
  - Es.: nome del database, ID e password di accesso, il driver ODBC da utilizzare
- L'insieme dei parametri necessari per connettersi ad una data fonte ODBC viene riferito per mezzo di un **DSN**
- Un DSN si definisce per mezzo del **Pannello di Controllo**

I DSN: tre tipi

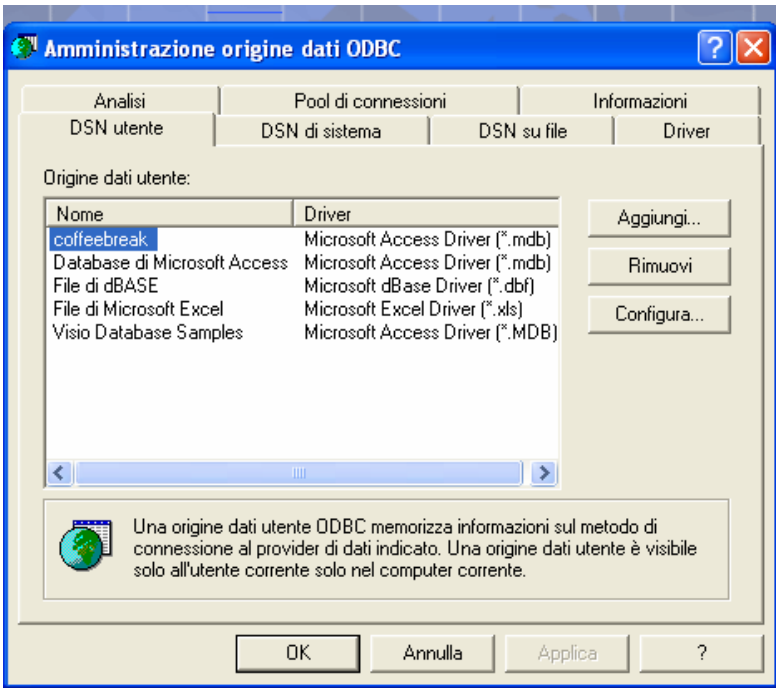
- **Le origini dati con DSN utente** sono locali rispetto a un computer e possono essere utilizzate soltanto dall'utente corrente.
- **Le origini dati con DSN di sistema** sono locali rispetto a un computer ma non sono dedicate a specifici utenti; qualsiasi utente con privilegi appropriati può infatti accedere a un DSN di sistema.
- **Le origini dati su file** possono essere condivise con altri utenti a condizione che abbiano installato gli stessi driver. Non è necessario che queste origini dati siano dedicate a un utente o siano locali rispetto a un particolare computer.

Connessione ad un database Access

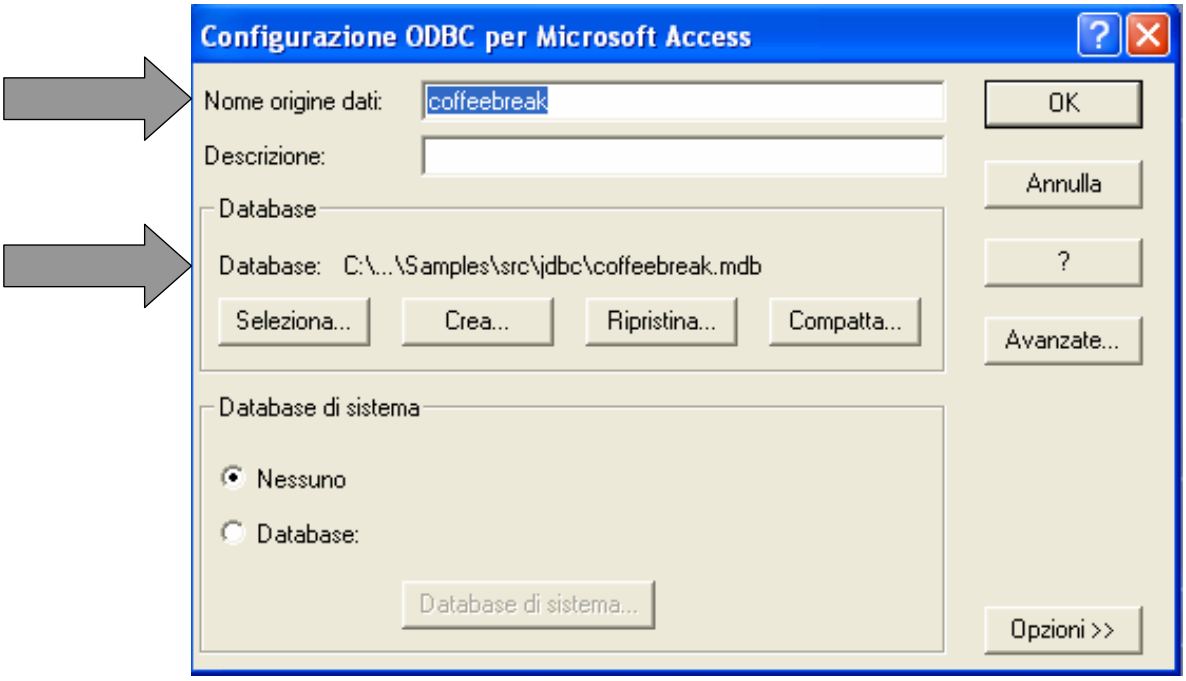
- **Realizzare (individuare) il database Access (.mdb)**
- **Creare un DSN ODBC di riferimento al database**
  - Windows XP: Pannello di Controllo – Strumenti di Amministrazione – Origine dati (ODBC)
    - ✓DSN Utente – Aggiungi
      - Nome origine dati
      - Database – Seleziona
- **Connettere il programma Java al database**

Aggiungi un DSN Utente (Access)

- Windows XP
- Pannello di Controllo
- Strumenti di Amministrazione
- Origine dati (ODBC)



Aggiungi un DSN Utente (Access)



Connessione ad un database MySQL

- **MYSQL database server & standard clients**
  - http://dev.mysql.com/downloads/
- **MYSQL Graphical clients**
  - Administrator
    - ✓http://dev.mysql.com/downloads/administrator/index.html
  - Query Browser
    - ✓http://dev.mysql.com/downloads/query-browser/index.html
- **JDBC DRIVER (Connector/j)**
  - http://dev.mysql.com/downloads/connector/j/3.1.html
  - mysql-connector-java-3.1.8a.zip
  - **mysql-connector-java-3.1.8-bin.jar**

Connessione ad un database MySQL

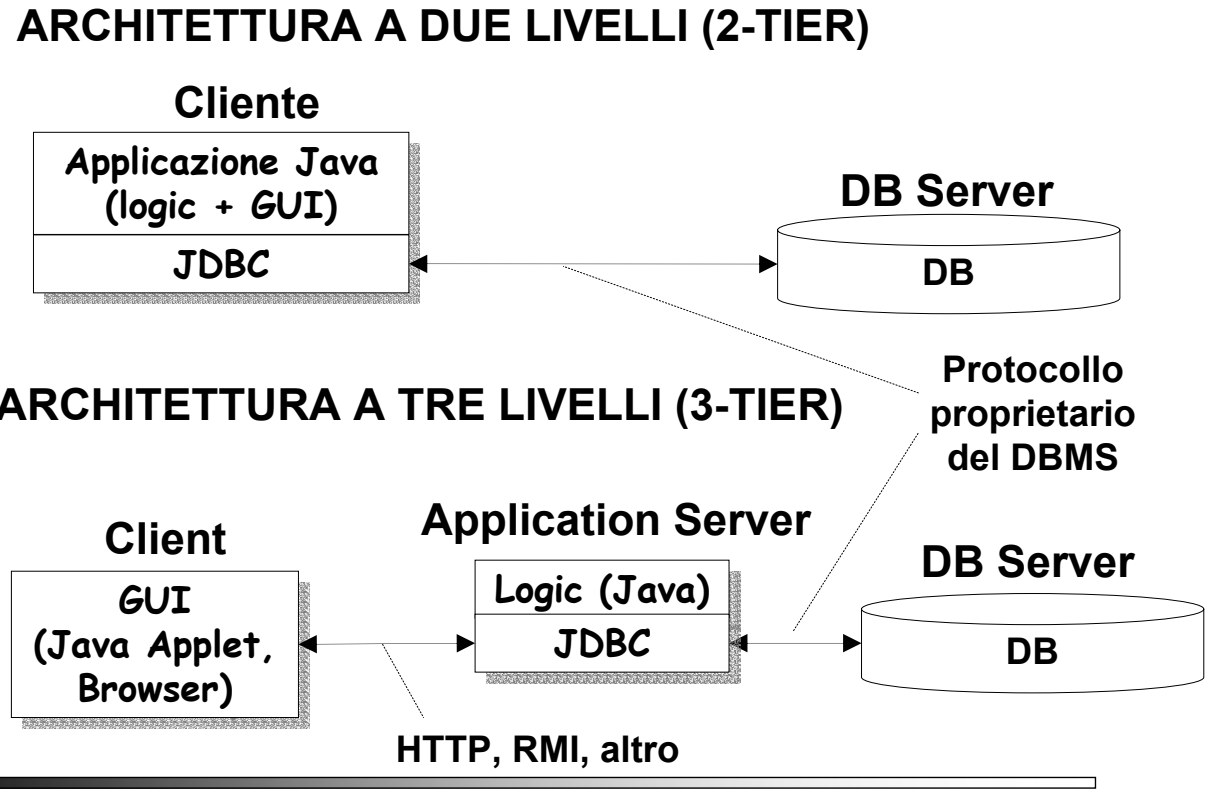
- Si assuma che il Connector/J sia posizionato nella cartella <folder>

```
import java.sql.*;
public class HelloMySQL {
   private static final String  DBMS_DRIVER = "org.gjt.mm.mysql.Driver";
   private static final String  DB_URL   = "jdbc:mysql:///localhost:3306/Banca?user=root&password=";

   public static void main(String[] args) {
      try {
         Class.forName(DBMS_DRIVER);
         Connection conn = DriverManager.getConnection(DB_URL);
         // …
      } catch (Exception e) {
         e.printStackTrace();
      }
}
```

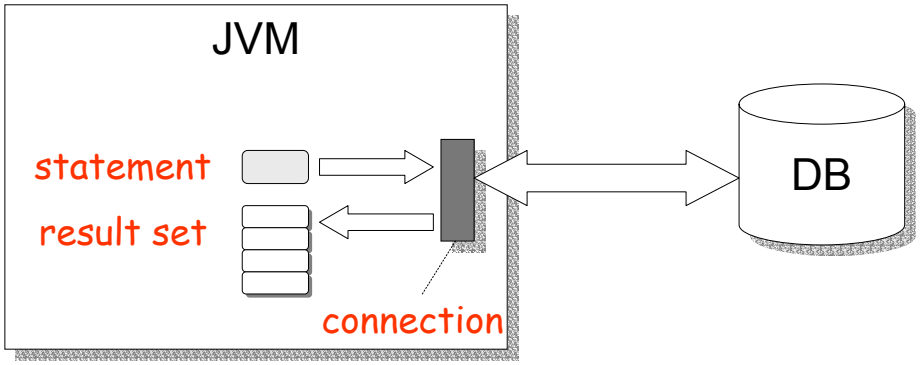
- set CLASSPATH=
  javac -classpath <folder>\mysql-connector-java-3.1.8-bin.jar; *.java
  java -classpath <folder>\mysql-connector-java-3.1.8-bin.jar; HelloMySQL

Architetture a 2 o più livelli



Passi per operare con un database

- Caricare e registrare il **driver**
  - Determinare il driver: *la classe è specificata dalla documentazione*
- Effettuare la **connessione**
- Inviare **statement** (comandi SQL)
- Ricevere **risultati** (un insieme di risultati)



Registrazione del driver

```
import java.sql.DriverManager;
import java.sql.Connection;

public class CaricaDriver {
   public static void main(String args[]) {
      String url = "jdbc:odbc:coffeebreak";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"));
         Connection con =
                  DriverManager.getConnection(url);
      }
      catch (Exception e)
      {
         System.out.println(e);
         e.printStackTrace();
      }
   }
}
```

- `Class.forName` registra il driver caricando la sua classe (il nome della classe è specificato dal produttore del driver)

  La registrazione del driver può avvenire anche per mezzo del meccanismo delle **proprietà**
- Con `getConnection` il Driver Manager cerca il driver opportuno fra quelli caricati
- Un **URL JDBC** ha il formato

  `jdbc:`*`subprotocol`*`:`*`subname`*

  dove il *`subprotocol`* specifica il driver e *`subname`* specifica il DB vero e proprio (il loro formato è specificato dal produttore)

Il database di riferimento

COFFEES

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

SUPPLIERS

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Creazione di una tabella

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;

public class CreateCoffee {
   public static void main(String args[])    {
      String createTableCoffee = "CREATE TABLE COFFEES " +
                                 "(COF_NAME VARCHAR(32), " +
                                 "SUP_ID INTEGER, PRICE FLOAT, " +
                                 "SALES INTEGER, TOTAL INTEGER)";
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"));
         Connection con =
              DriverManager.getConnection("jdbc:odbc:coffeebreak");
         Statement stmt = con.createStatement();
         stmt.executeUpdate(createTableCoffee);
         con.close();
      }
      catch (Exception e)
      {
         System.out.println(e);
         e.printStackTrace();
      }
   }
}
```

- Il comando SQL non termina con alcun terminatore: è il driver che ci inserisce quello appropriato
- **executeUpdate** permette di eseguire **DDL statement** (**create**, **alter**, **drop**) e **query di comando** (**update, insert**)

Inserimento dei dati di una tabella

```
Statement stm = con.createStatement();
stmt.executeUpdate("INSERT INTO COFFEES " +
                   "VALUES ('Colombian', 101, 7.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
                   "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
                   "VALUES ('Colombian_Decaf', 101, 8.99, " +
                                        "0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
                   "VALUES ('French_Roast_Decaf', 49, 9.99, " +
                   "0, 0)");
con.close();
```


Interrogazione di una tabella

```
String query = "SELECT * FROM COFFEES";
…
try {
   // connessione al database
   Statement stmt = con.createStatement();

   ResultSet rs = stmt.executeQuery(query);
   while ( rs.next() ) {
      String s = rs.getString("COF_NAME");
      int n = rs.getInt("SALES");
      System.out.println(n + " kg di " + s + " questa settimana.");
   }
} catch (Exception e) {
   /* gestione */
}
```

- Un oggetto **ResultSet** è, concettualmente, una tabella che rappresenta il risultato di una query
- Un oggetto **ResultSet** mantiene un puntatore alla riga corrente; inizialmente il puntatore è posizionato prima della prima riga
- Un **ResultSet** non è né **aggiornabile** né **scrollabile** (il puntatore si muove solo in avanti)
- In **getXXX** è possibile specificare il nome, o l'indice della colonna (del risulato)

Prepared Statements

Prepared statement permettono di definire **statement parametrici (query di aggiornamento e di selezione)**

```
PreparedStatement updateSales;
String updateString = "UPDATE COFFEES SET SALES = ?" +
                      "WHERE COF_NAME LIKE ?";
updateSales = con.prepareStatement(updateString);
int[] salesForWeek = {175, 150, 60, 155, 90};
String[] coffees = {"Colombian", "French_Roast", "Espresso",
                    "Colombian_Decaf", "French_Roast_Decaf"};
for (int i = 0; i < coffees.length; i++) {
   updateSales.setInt(1, salesForWeek[i]);
   updateSales.setString(2, coffees[i]);
   updateSales.executeUpdate();
}
```

- **setXXX** permette di specificare un valore attuale per il parametro
- Un parametro mantiene il valore specificato a meno che non ne venga specificato un altro
- Se un prepared statement **può essere riutilizzato**: è sufficiente specificare i valori che cambiano tra una query e l'altra

Join

```
String join = "select COF_NAME " +
              "from COFFEES, SUPPLIERS " +
              "where SUP_NAME LIKE 'Acme, Inc.' " +
              "and SUPPLIERS.SUP_ID = COFFEES.SUP_ID";
try {
  // connessione
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery(join);
  while ( rs.next() ) {
     String s = rs.getString("COF_NAME");
     System.out.println(s);
  }
} catch (Exception e) {
   /* gestione }
```

Transazioni

- Una **transazione** è un insieme di operazioni che devono essere eseguite in maniera **atomica**, cioè o **tutte** le operazioni vengono eseguite con successo oppure **nessuna** operazione viene eseguita
- Il comando SQL **COMMIT** fa terminare con successo una transazione
- Il comando SQL **ROLLBACK** fa abortire o terminare senza successo la transazione

Transazioni JDBC

- Una connessione è creata in **auto-commit mode**
- Ogni comando SQL viene considerato come una transazione ed automaticamente "committato"
- Per eseguire una **transazione costituita da più comandi** bisogna
  - **disabilitare la modalità auto-commit** (**con.setAutoCommit(false)**)
  - **eseguire i comandi**
  - **invocare esplicitamente il comando di COMMIT** (**con.commit()**)
  - **riabilitare la modalità auto-commit** (**con.setAutoCommit(true)**)

Esempio di transazione con più comandi

```
try {
  PreparedStatement updateSales = con.prepareStatement(
     "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
  PreparedStatement updateTotal = con.prepareStatement(
     "UPDATE COFFEES SET TOTAL = TOTAl + ? WHERE COF_NAME LIKE ?");
  int [] salesForWeek = {175, 150, 60, 155, 90};
  String [] coffees = {"Colombian", "French_Roast", "Espresso",
              "Colombian_Decaf", "French_Roast_Decaf"};
  int len = coffees.length;
  con.setAutoCommit(false); // disabilita auto-commit
  for (int i = 0; i < len; i++) {
     updateSales.setInt(1, salesForWeek[i]);
     updateSales.setString(2, coffees[i]);
     updateSales.executeUpdate(); // primo statement
     updateTotal.setInt(1, salesForWeek[i]);
     updateTotal.setString(2, coffees[i]);
     updateTotal.executeUpdate(); // secondo statement
     con.commit(); // commit di una riga
  }
  con.setAutoCommit(true); // riabilita auto-commit
  updateSales.close();
  updateTotal.close();
} catch(SQLException e) {/* vedere dopo */}
```

L'auto-commit deve essere disabilitato solo per la durata della transazione

Rollback

Se l'esecuzione di uno più statement non ha successo viene lanciata un eccezione di tipo **SQLException**

Tuttavia, **non c'è modo di sapere cosa è andato storto** perció bisogna:

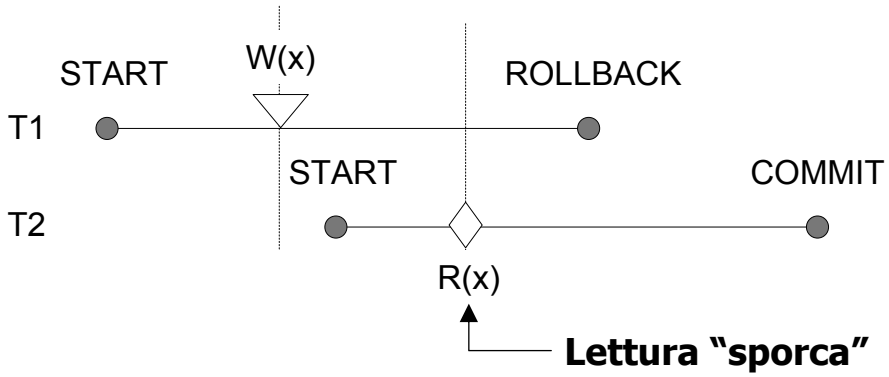
- **intercettare l'eccezione** (clausola **catch()**)
- **abortire la transazione** (**con.rollback()**)
- eventualmente ritentare l'esecuzione della transazione (ad esempio, inserendo la transazione in un ciclo)

Esempio di rollback

```
try {
   // connessione con il database
   // transazione
} catch(SQLException e) {
   System.err.println("SQLException: " + e.getMessage());
   if (con != null) {
      try {
         System.err.print("Transaction is being rolled back");
         con.rollback();
      } catch(SQLException ex) {
         System.err.print("SQLException: " + ex.getMessage());
      }
   }
}
```

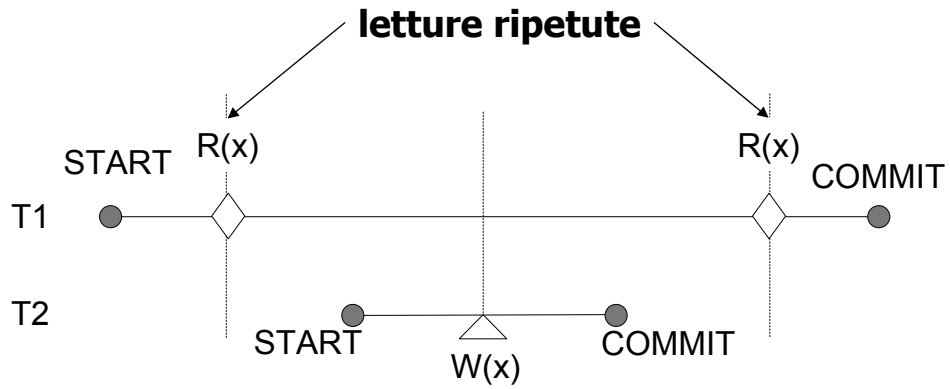

Transazioni concorrenti: dirty read

x: record
W: scrittura
R: lettura
T1 || T2



- La transazione T2 legge un record "sporco" (non committed) che non farà mai parte del data base
- Esempio: x specifica l'indirizzo di un cliente

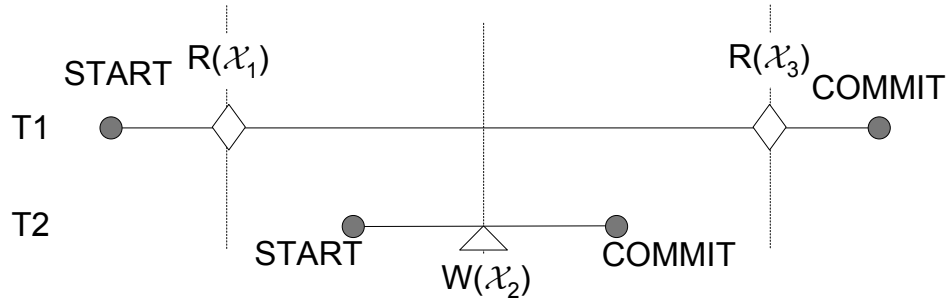
Transazioni concorrenti: repeatable read



x: record
W: scrittura
R: lettura
T1 || T2

- T1 legge un valore diverso di x in seconda lettura
- Esempio: x contiene la data di scadenza di un prodotto specifico che viene scritta da T1 su un certo numero di etichette

Transazioni concorrenti: phantom read



x: record
W: scrittura
R: lettura
T1 || T2

W($\mathcal{X}_2$) inserisce record “fantasma”

- $\mathcal{X}_i$ insieme di record che soddisfano una certa condizione $\mathcal{C}$
- Ad esempio, $\mathcal{X}_3 = \mathcal{X}_1 \cup \mathcal{X}_2$
- T1 legge recod “fantasma” in seconda lettura
- La lettura di record fantasma viola la serializzabilità ma, in pratica, non crea effetti collaterali “devastanti”

Livello di isolamento

Il **livello di isolamento** di una transazione $\mathcal{T}$ determina la "visibilità" che $\mathcal{T}$ ha delle transazioni concorrenti

Lo standard SQL-ANSI definisce **quattro livelli**, ciascuno dei quali realizza **un diverso compromesso tra consistenza e prestazioni**

- **READ UNCOMMITTED**: permette le letture sporche, le letture ripetibili e le letture fantasma; massime prestazioni
- **READ COMMITTED**: non permette le letture sporche; comportamento di default
- **REPEATABLE READ**: permette solo le letture fantasma
- **SERIALIZABLE**: totale isolamento; serializza le transazioni

Livelli di isolamento in JDBC

- L'interfaccia **Connection** fornisce i metodi **void setTransactionIsolation(int level)**, **void getTransactionIsolation(int level)** dove **level** può assumere un valore costante corrispondente ai livelli standard
- **N.B.:** Anche se JDBC permette di impostare un livello di isolamento, l'effettiva soddisfazione del livello dipende dal driver e dal DBMS sottostante

Corrispondenza tra i tipi

Tipo SQL	Tipo Java
`INTEGER o INT`	`int`
`SMALLINT`	`short`
`NUMERIC(m, n), DECIMAL(m, n) o DEC(m, n)`	`java.sql.Numeric`
`FLOAT (n)`	`double`
`REAL`	`float`
`DOUBLE`	`double`
`CHARACTER(n) or CHAR(n)`	`String`
`VARCHAR(n)`	`String`
`BOOLEAN`	`Boolean`
`DATE`	`java.sql.Date`
`TIME`	`java.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp`
`BLOB`	`java.sql.Blob`
`CLOB`	`java.sql.Clob`
`ARRAY`	`java.sql.Array`

BLOB/CLOB (Binary and Character Large Object)

Eccezioni ed Warning

- Un'eccezione **e** di tipo **SQLException** ha tre componenti
  - **messaggio** ottenuto con **e.getMessage()**
  - **stato SQL** ottenuto con **e.getSQLState()** in accordo con X/Open SQLState
  - **codice di errore** (proprietario) ottenuto con **e.getErrorCode()**
- **Una o più eccezioni di tipo SQLException possono essere concatenate** per fornire informazioni addizionali sull'errore
- Una **SQLWarning** è una sottoclasse di **SQLException** che gestisce le warning di accesso al database (una warning non fa terminare l'esecuzione)

Gestione eccezioni

```
try {
   // Codice che può generare un'eccezione.
   // Se un'eccezione viene generata il blocco catch sotto
   // stampa le informazioni su di essa.
} catch(SQLException ex) {
   System.out.println("\n--- SQLException catturata ---\n");
   while (ex != null) {
      System.out.println("Message: " + ex.getMessage());
      System.out.println("SQLState: " + ex.getSQLState());
      System.out.println("ErrorCode: " + ex.getErrorCode());
      ex = ex.getNextException();
      System.out.println("");
   }
}
```


Result Set in JDBC2

- **Scrollable**: è possibile spostarsi avanti ed indietro e saltare ad una particolare riga nel RS
- **Updatable (1)**: se il contenuto del RS viene modificato, le modifiche sono propagate al DB
- **Updatable (2)**: il RS viene aggiornato se una una transazione concorrente modifica i dati

Scrollable Result Set (JDBC 2)

Statement stat = con.createStatement(type, concurrency)

TYPE

- **TYPE_FORWARD_ONLY**: il RS non è "scrollable"
- **TYPE_SCROLL_INSENSITIVE**: il RS è "scrollable" ma non è sensibile alle modifiche nel DB
- **TYPE_SCROLL_SENSITIVE**: il RS è "scrollable" e sensibile alle modifiche nel DB

CONCURRENCY

- **CONCUR_READ_ONLY**: il RS non può essere utilizzato per modificare il DB
- **CONCUR_UPDATABLE**: il RS può essere utilizzato per modificare il DB

Result Set updtable

```
String query = "SELECT * FROM COFFEES";
double increase = 0.1;
try  {
   // connessione al database
   Statement stat = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
   ResultSet rs = stat.executeQuery(query);
   while ( rs.next() ) {
      String cof_name = rs.getString("COF_NAME");
      double price = rs.getDouble("PRICE");
      double newPrice = price * (1 + increase);
      rs.updateString("COF_NAME", cof_name);
      rs.updateDouble("PRICE", newPrice);
      rs.updateRow();
   }
   if ( rs.first() )
      do
         System.out.println(rs.getString("COF_NAME") + ": " +
                            rs.getDouble("PRICE"));
      while ( rs.next() );
   stat.close();
   con.close();
} catch(SQLException e) {/* gestione */}
```

Result Set updatable: inserimento di un record

```
String query = "SELECT * FROM COFFEES";
try  {
   // connessione al database
   Statement stat = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
   ResultSet rs = stat.executeQuery(query);
   rs.moveToInsertRow(); // sposta il cursore sulla "riga di inserimento"
   rs.updateString("COF_NAME", "Ticos");
   rs.updateInt("SUP_ID", 150);
   rs.updateDouble("PRICE", 8.99);
   rs.updateInt("SALES", 0);
   rs.updateInt("TOTAL", 0);
   rs.insertRow(); // invia le modifiche al al datadase
   // chiusura statement e connessione
} catch(SQLException e) {/* gestione */}
```

Il metodo **moveToCurrentRow()** riporta il cursore del RS sulla riga corrente

Result Set Updatable: rimozione di un record

```
try  {
   // caricamento della classe driver
   // connessione al database
   Statement stat = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
   ResultSet rs = stat.executeQuery(query);
   while ( rs.next() ) {
     String cof_name = rs.getString("COF_NAME");
      if ( "Espresso".equals(cof_name) ) {
         rs.deleteRow(); // elimina la riga corrente
         break;
      }
   }
   stat.close();
   con.close();
} catch (Exception e) {
```

Scrollable & Updatable RS: commenti

- **L'utilizzo di RS updatable può dare luogo a programmi inefficienti**
  - RS updatable sono da utilizzarsi in programmi interattivi, altrimenti
  - è da preferirsi l'utilizzo di UPDATE
- **Un DB può non onorare una richiesta per RS updatable e scrollable**. Ci sono vari modi per accertarsene:
  - **DatabaseMetaData.supportsResultSetType**, **DatabaseMetaData.supportsResultSetConcurrency**
  - Warning
  - **ResultSet.getType**, **ResultSet.getConcurrency**

Metadata

I dati che descrivono la struttura di un DB sono detti metadati

Ad esempio: il nome delle tabelle, nome e tipo delle colonne di una tabella

Ci sono due tipi di metadati

- **I metadati delle tabelle**: **DataBaseMetaData**
- **I metadati dei result set**: **ResultSetMetaData**

Metadata: esempio

```
try  {
   // caricamento della classe driver
   // connessione al database
   DatabaseMetaData meta = conn.getMetaData();
   ResultSet rs = meta.getTables(null, null, null,
                                 new String[]{"TABLE"});
   while ( rs.next() )
      System.out.println(rs.getString(3));
   rs.close();

    String query = "SELECT * FROM COFFEES";
    Statement stat = conn.createStatement();
    rs = stat.executeQuery(query);
    ResultSetMetaData rsmeta = rs.getMetaData();
    for ( int i = 1; i < rsmeta.getColumnCount(); i++ ) {
       String columnName = rsmeta.getColumnLabel(i);
       int columnWidth = rsmeta.getColumnDisplaySize(i);
       System.out.println(columnName + ", " + columnWidth);
    }
    rs.close();
    conn.close();
} catch (Exception e) { /* gestione eccezioni */ }
```


JDBC in Web Application

- **Scrollable Result Set** richiede una connessione costante con il DB.

  In applicazioni Web è preferibile utilizzare **RowSet** (**extends ResultSet**) che non ha questo requisito

- **Pool di connessioni**. Invece di creare e distruggere una connessione dinamicamente, un programma si fa allocare temporaneamente una connessione da un pool di connessioni disponibili

  Tipicamente è supportata dai produttori di Application Server (JBoss, BEA, WebLogic, IBM WebSphere)